## 领域服务

### 聚合的三个问题

按照面向对象设计原则，需要将“ **数据与行为封装在一起** ”，避免将领域模型对象设计为贫血对象。如此一来，聚合内的实体与值对象承担了与其数据相关的领域行为逻辑。聚合满足了领域概念的完整性、独立性与不变量，体现了聚合才是对象粒度之上自治单元的最佳选择。一个设计良好的聚合同样需要做到最小完备、自我履行、稳定空间和独立进化。

在领域驱动设计中，自治对象参与了聚合内部的协作，而聚合作为多个实体与值对象的整体，则是参与业务场景的自治单元。倘若将聚合拥有的数据称为“已知数据”，若一个业务场景的领域行为只需操作这些已知数据，就应定义为聚合内相关类的领域方法，倘若该领域行为还需要和别的聚合协作，该方法就被分配给聚合根实体。有时候，聚合的已知数据并不足以支持整个业务场景的领域需求，为了保证聚合的自治性，会将不足的部分通过聚合方法的参数传入。通过参数传入的外部数据皆可视为聚合的“未知数据”。

**为了彰显聚合边界的防御作用，聚合并不会直接与别的聚合协作，那么这些未知数据从何而来？**

聚合封装了多个实体和值对象，聚合根是访问聚合的唯一入口。若要遵循迪米特法则，聚合调用者应该只需知道暴露在外的聚合根实体，当业务需求需要调用聚合内实体或值对象的方法时，聚合当隐去其细节，由根实体包装这些方法，在方法内部的实现中，将外部的请求委派给内部的相应类。封装的领域行为被固化在聚合之中，成为了聚合的附庸。这是将对象作为一等公民的面向对象设计思想的体现。

**在业务系统中，总会有一些领域行为游离在聚合之外，它们要么不需要聚合自身携带的已知数据，要么存在与聚合迥然不同的变化方向，这些领域行为应该附庸在哪个对象之上呢？**

作为自治单元的聚合是领域层的主力军，在企业软件系统中，它封装了系统最为核心的业务功能，在整洁架构思想中，又被视为系统最为稳定的领域模型。从分离变与不变的设计思想来看，聚合必然不能与访问外部资源的技术实现混合在一处；即使领域模型因业务需求而变，但它与外部资源的变化方向定然不同，故而亦当完全分离。无论外部资源是数据库、消息队列还是网络通信，都可以通过抽象的南向网关（包括资源库），作为领域逻辑与技术实现的分水岭。

**不管聚合做到了多大程度的自治，总需要与抽象的南向网关协作，如此才能实现完整的业务场景，这样的协作行为如果不能分配给聚合，又该分配给什么对象呢？**

以上三个问题汇成一个答案，曰： **领域服务** 。

### 什么是领域服务

“服务”这个词语在软件领域中实在过于泛滥，而它的宽泛性使得我们甚至无法给出一个准确的定义。服务的特征却显而易见：

* 首先，服务并不是某一个具体的事物（thing）
* 其次，服务体现的是一种行为（behavior）

故而，领域服务（Domain Service）代表了在名词世界（面向对象）中对动词的封装（接口），它封装了领域行为。前提在于，这一领域行为在实体或值对象中找不到容身之处。换言之，当我们针对领域行为建模时，需要优先考虑使用值对象和实体来封装领域行为，只有确定无法寻觅到合适的对象来承担时，才将该行为建模为领域服务的方法。

虽然领域服务是领域设计建模的最末选择，但“服务”这个词语实在太过宽泛了，在表达业务逻辑时，只要服务的修饰语合适，就有充足的理由将相关的领域逻辑分配给它。如此就会导致领域服务的泛滥，成为一个无所不包的“上帝”服务。长此以往，所有的业务逻辑都会放到这个服务中，使得领域层的设计重新走回“贫血模型”的老路。

如果阅读供应链的开源项目[OFBiz](https://ofbiz.apache.org/)，这种贫血模型加上帝服务的形式就处处可见。例如，该项目定义了 ShipmentService 服务，该服务类包含了一千多行代码，服务定义的公开方法（坦白说，这些方法的命名很好地体现了领域特征）包括：

* createShipmentEstimate()
* removeShipmentEstimate()
* calcShipmentCostEstimate()
* fillShipmentStagingTables()
* updateShipmentsFromStaging()
* clearShipmentStagingInfo()
* updatePurchaseShipmentFromReceipt()
* duplicateShipmentRouteSegment()
* quickScheduleShipmentRouteSegment()
* getShipmentPackageValueFromOrders()
* sendShipmentCompleteNotification()
* getShipmentGatewayConfigFromShipment()

这就是典型的“上帝”服务，庞大和臃肿，严重违背了单一职责原则。表面是服务对象，其实每个服务方法都是一个事务脚本，缺乏内聚职责的封装，也缺乏对领域模型概念的呈现，成为一种彻头彻尾的过程式实现。若要从领域建模的角度分析，仅需从该服务诸多方法的命名，也可察觉领域概念的端倪。例如：

* 与运输费用估算有关：ShipmentCostEstimate
* 与分段运输有关：ShipmentStaging
* 与运输路径有关：ShipmentRouteSegment
* 与运输包有关：ShipmentPackage
* 与运输收据有关：ShipmentReceipt

通过这些领域行为甄别出来的领域概念，完全可以定义为相关的实体或值对象，由其承担一部分与其数据有关的领域行为。为何会出现这样的实现呢？我想问题还是在于服务概念的过宽过泛。定义的服务 ShipmentService，其言外之意，凡是与运输（Shipment）有关的业务，或多或少都会与该服务扯上关系。软件的设计人员与开发人员往往存在一种惰性，不愿意锱铢必较探究职责分配的合理性，一旦认为该业务与运输有关，就自然而然考虑分配给 ShipmentService，然后再借助过程式思维模式，按照结构顺序编写代码，就形成了我们看到的事务脚本。

随着业务的逐渐增加，这种看似理所当然的职责分配就会让整个服务陷入庞大臃肿的泥沼之中。显然，如果在设计与开发时对职责的分配不加约束，所谓的“职责分治”不过是一句空话罢了。为了避免这种现象，在对领域进行建模时，考虑设计要素的顺序应该为：

值对象（Value Object）→ 实体（Entity）→ 领域服务（Domain Service）

为了避免开发人员把领域服务当做一个“筐”，什么逻辑都往里面装，除了需要提高团队成员面向对象的设计能力，强调领域建模的设计顺序之外，还有一个方法，就是对领域服务加以约束。可惜的是，没有任何语言可以施加领域驱动设计要素的约束。Mat Wall 与 Nik Silver 在 Guardian.co.uk 网站推进领域驱动设计时的实践值得我们借鉴。他们在文章《[演进架构中的领域驱动设计](http://www.infoq.com/cn/articles/ddd-%20evolving-architecture)》中建议：

为了对付这一行为，我们对应用中的所有服务进行了代码评审，并进行重构，将逻辑移到适当的领域对象中。我们还制定了一个新的规则： **任何服务对象在其名称中必须包含一个动词。** 这一简单的规则阻止了开发人员去创建类似于 ArticleService 的类。取而代之，我们创建 ArticlePublishingService 和 ArticleDeletionService 这样的类。推动这一简单的命名规范的确帮助我们将领域逻辑移到了正确的地方，但我们仍要求对服务进行定期的代码评审，以确保我们在正轨上，以及对领域的建模接近于实际的业务观点。

通过限制服务命名来规范领域模型的设计，看似荒唐，其实真如天外飞仙，颇有创见，因为它实则体现了领域服务的行为本质。这个行为是无状态的，相当于一个纯函数。发布文章是一个领域行为，对应于 publishArticle() 函数；删除文章是一个领域行为，对应于 deleteArticle() 函数。只是在 Java 中，无法直接定义这样的函数，不得已才定义类或接口作为函数“附身”的类型罢了。

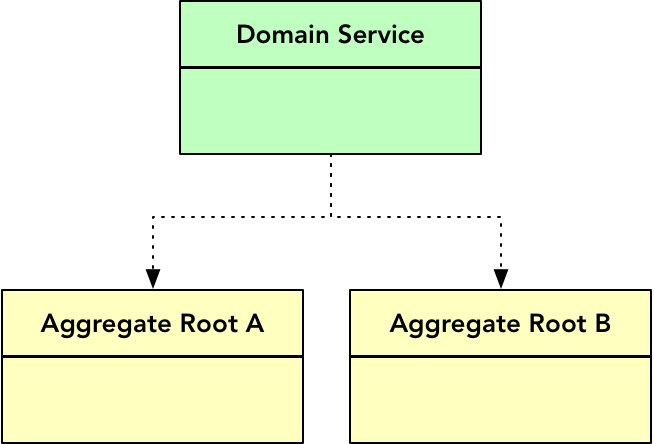
命名约束的实践可能会导致太多细粒度的领域服务产生，但在领域层，这样的细粒度设计值得提倡，它能促进类的单一职责，保证类的重用和应对变化的能力。由于每个服务的粒度非常细，就不可能产生包罗万象的“上帝”服务。服务的定义是有设计成本的。在创建一个新的领域服务时，命名约束为让我们暂时停下来，想一想，要分配给这个新服务的领域逻辑是否有更好的去处？

### 领域服务的应用场景

领域服务不只限于对领域行为的建模，在领域设计模型中，它与聚合、资源库等设计要素拥有对等的地位。领域服务的应用场景是有设计诉求的，恰好可以呼应前面提及的三个问题。

**第一个问题：** 为了彰显聚合边界的防御作用，聚合并不会直接与别的聚合协作，那么这些未知数据从何而来？

多数时候，一个自治的聚合无法完成一个完整的业务场景，需要共同协作才能完成。然而，聚合的设计原则却要求聚合之间只能通过聚合的身份标识进行协作。这就意味着在聚合之上，需要引入一个设计对象来封装这种聚合之间的协作行为。这就是领域服务承担的职责：



49820588.png

然则领域服务又是从何处获得聚合对象的呢？一个可能是领域服务的调用者传递给它。领域服务的调用者可以是另外一个领域服务，但在多数情况下应为应用服务。应用服务的调用者又为远程服务，最终为发起服务请求的前端或第三方服务。我在《领域驱动分层架构与对象模型》一节中将客户端发送的请求消息分为查询消息与命令消息。对于聚合而言，客户端的请求消息最终会到达领域服务，据不同的操作类型转换为不同的参数，与管理聚合生命周期的对象进行协作：

* 查询操作：请求消息为查询条件，由资源库根据查询条件获得聚合对象
* 创建操作：命令消息作为输入参数，由工厂负责创建聚合对象，然后由资源库执行新增操作
* 更新操作：命令消息中必含有聚合的身份标识，由资源库根据身份标识获得聚合对象，再根据命令消息的新值由聚合对象在内存中更新其状态，最终由资源库执行更新操作
* 删除操作：命令消息包含查询条件，由资源库根据查询条件执行删除操作

不管是什么操作，与领域服务协作的聚合对象都不是与生俱来的，也不是通过外部调用者传递而来，而是通过工厂或资源库创建或获取而来。因此，领域服务在协调多个聚合之间的协作时，还需要与工厂或资源库协作。

例如，针对“验证订单有效性”这一验证行为，需要验证订单自身属性的完备性，包括验证订单是否提供了配送地址、联系人信息，还要确保订单聚合的不变量，如保证订单包含了有效的订单项。这些信息皆属于 Order 聚合边界内各个类的属性，基于聚合的自治原则，将由 Order 聚合自身来承担验证功能。在验证订单有效性时，还需要验证下订单的顾客是否为有效顾客。顾客是另一个聚合 Customer 的根实体，这就牵涉到两个聚合之间的协作。故而需要引入领域服务 ValidatingOrderService。该领域服务封装了“验证订单”这一领域行为，需要传入被验证的 Order 聚合对象。由于聚合之间的协作只能通过身份标识进行，Order 聚合没有引用 Customer 聚合，而是持有顾客的身份标识 CustomerId。要获得 Customer 聚合，就需要通过该聚合的资源库：

public class ValidatingOrderService {  
 private CustomerRepository customerRepo;  
  
 public boolean isValid(Order order) {  
 try {  
 order.validate();  
  
 Optional<Customer> optCustomer = customerRepo.customerOf(order.getCustomerId());  
 return optCustomer.isPresent();  
 } catch (InvalidOrderExceptiion ex) {  
 log.info(ex.getMessage());  
 return false;  
 }  
 }  
}

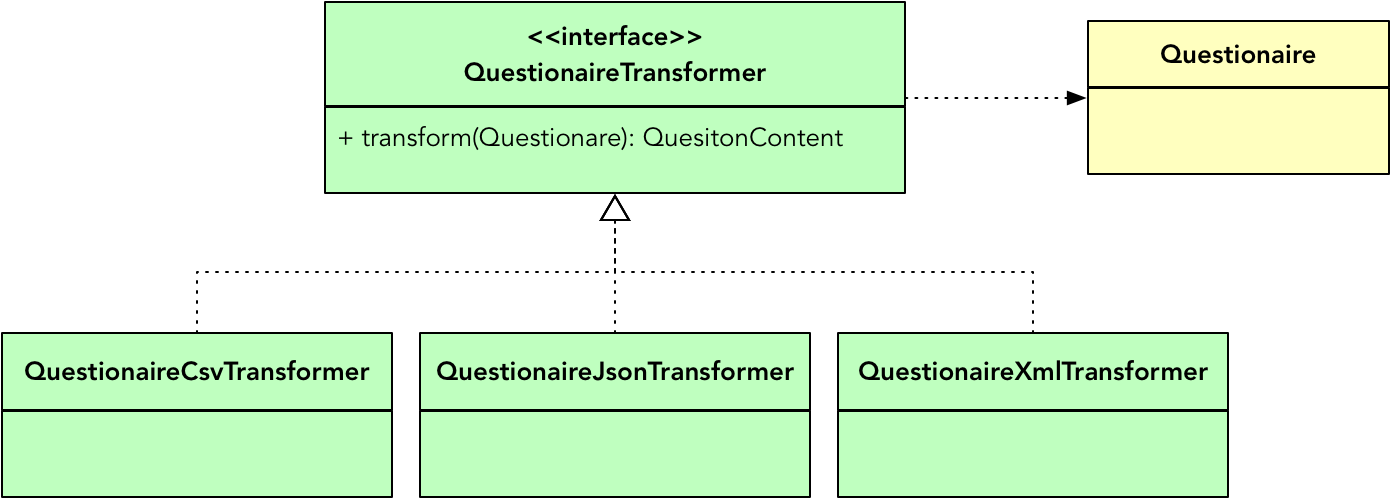
**第二个问题** ：在业务系统中，总会有一些领域行为游离在聚合之外，它们要么不需要聚合自身携带的已知数据，要么存在与聚合迥然不同的变化方向，这些领域行为应该附庸在哪个对象之上呢？

设计时，我们首先需要遵循“数据与行为封装在一起”的设计原则。有时候，行为的变化方向却与拥有数据的类并非一致，这时就应分离变与不变，将这一变化的领域行为从它所属的聚合中剥离出来，形成领域服务。由于领域行为存在变化，为了满足扩展要求，还应在领域服务基础上建立抽象。许多行为型设计模式，如策略模式（Strategy Pattern）、命令模式（Command Pattern）、访问者模式（Visitor Pattern）都采用了分离并抽象行为的设计。如果这些行为与领域逻辑有关，则抽象的策略接口、命令接口、访问者接口都可以视为领域服务。

例如，保险系统常常需要客户填写一系列问卷调查，以了解客户的具体情况，从而确定符合客户需求的保单策略。调查问卷 Questionaire 是一个聚合根实体，其内部是由多个处于不同层级的值对象组成的树形结构：

Section ->  
 SubSection ->  
 QuestionGroup->  
 Question->  
 PrimitiveQuestionField

业务需求要求将一个完整的调查问卷导出为多种形式的文件，这就需要提供转换行为，将一个聚合的值转换为多种不同格式的内容，例如 CSV 格式、JSON 格式与 XML 格式。转换行为操作的数据为 Questionaire 聚合所拥有，若按照数据与行为应封装在一起的原则，该行为代表的职责就应该由聚合自身来履行。然而，这个转换行为却存在多种变化，不同的内容格式代表了不同的实现。正确的做法就是将转换行为从 Questionaire 聚合中分开，并建立一个抽象的接口 QuestionaireTransformer：



76979171.png

**第三个问题** ：不管聚合做到了多大程度的自治，总需要与抽象的南向网关协作，如此才能实现完整的业务场景，这样的协作行为如果不能分配给聚合，又该分配给什么对象呢？

领域逻辑要做到纯粹地不依赖任何外部资源，在真实的企业业务系统中，几乎不可能。我们只能建立不同粒度的领域模型对象，保证较小粒度的领域模型对象能够做到领域逻辑的纯粹性，在领域驱动设计中，这个粒度就是聚合。一旦领域行为突破了聚合粒度，就很有可能牵涉到与外部资源的协作。在领域层，可以将所有的外部资源都视为一个抽象的网关（Gateway），其中，资源库是一种特殊的针对数据库的网关。

领域服务在协调多个聚合的协作时，由于聚合协作关系的限制，必须引入资源库参与协作。这一点在解释第一个问题时，已经提及。不仅限于此，即使领域行为仅仅操作一个聚合，只要它还需要与外部资源交互，那么这一职责就应该交由领域服务来承担。这实际上遵循了领域驱动设计中的一个原则： **应该尽量避免在聚合中使用资源库** 。

资源库是用来管理聚合的生命周期的，如果在聚合内部使用资源库，就意味着资源库在“重建”聚合根对象时，还需要将该聚合根对象依赖的资源库对象提供给它。遵循整洁架构思想，需要抽象资源库与依赖注入相结合才能避免内部领域层依赖外部资源层。当资源库实现通过 ORM 框架在数据库中获得聚合根对象时，依赖注入框架无法做到将资源库自身设值给聚合根。倘若聚合内部还使用了其他资源库，就更无法满足正常构建聚合对象的需求了。因此，在聚合中使用资源库，颇有几分像是蛋生鸡还是鸡生蛋的循环问题。

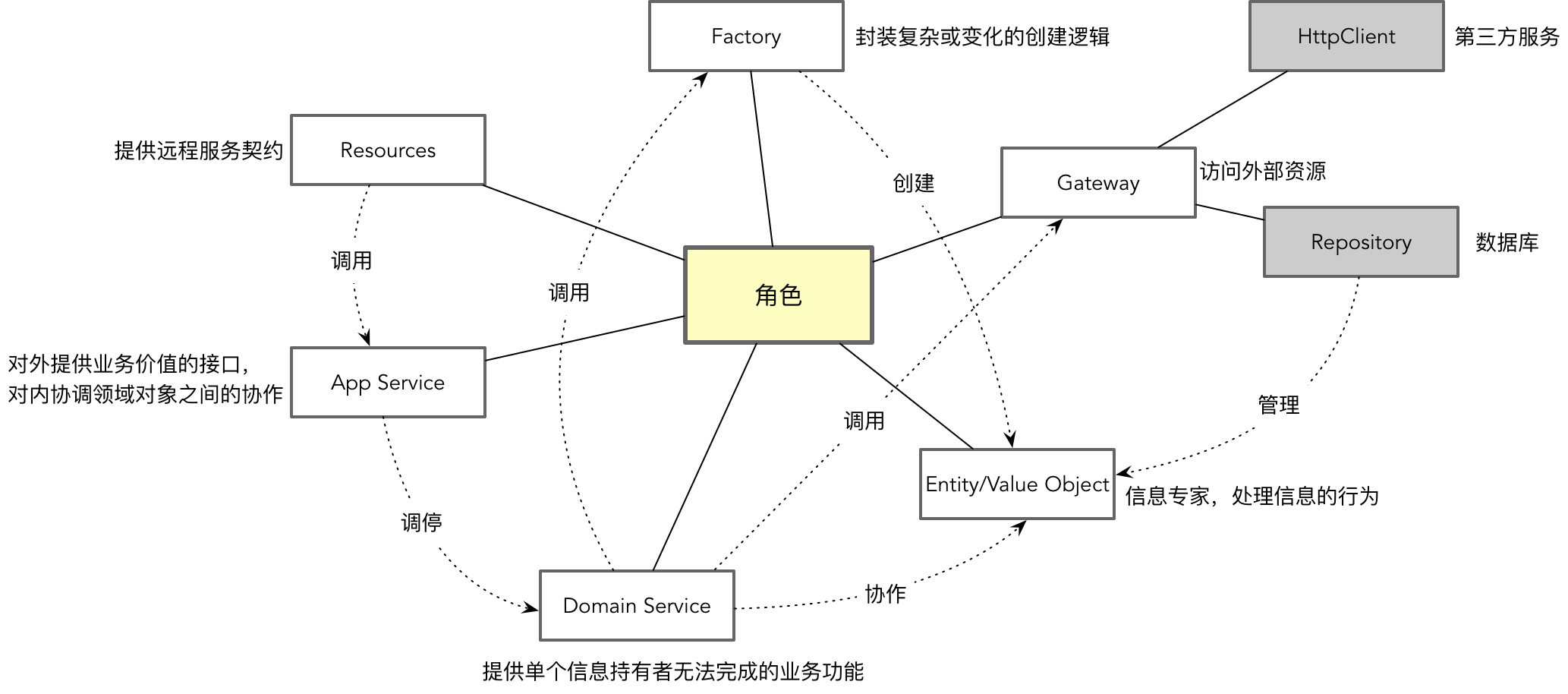
由于领域服务的生命周期并不需要资源库来管理，因此将调用资源库的职责转移到领域服务，该问题就能迎刃而解了。

以物流系统的合同管理功能为例。在创建合同时，需要用户为合同提供一个自编码。在用户输入自编码时，除了要验证该自编码是否满足编码规则之外，还要检测它在已有合同中是否已经存在。根据 **信息专家模式** ，拥有信息（自编码数据）的对象就是操作该信息的专家，如此一来，验证自编码的行为就应该分配给合同 Contract 聚合内的值对象 ContractNumber。但是，检测自编码是否已经存在，又需要访问外部的数据库，这又是聚合对象自身力有未逮的。故而，自编码的整体验证功能将交由领域服务，在其内部，又进行了职责的分解，形成多个对象角色之间的协作：

public class CustomizedNumberValidator {  
 private ContractRepository contractRepo;  
  
 public boolean isValid(CustomizedNumber number) {  
 try {  
 number.validate();  
 return !contractRepo.isDuplicatedNumber(number.value());  
 } catch (InvalidCustomizedNumberException ex) {  
 log.info(ex.getMessage());  
 return false;  
 }  
 }  
}

因为要访问外部资源，所以应该将该领域行为分配给领域服务，这可能导致细粒度的领域服务。然而，细粒度的领域服务有利于业务功能的重用，也能够更好地应对需求的变化。例如，创建合同和更新合同都会使用验证自编码合法性的功能，当自编码验证规则发生变化时，通过单独分离出来的 CustomizedNumberValidator 服务也可以更好地控制变化。

显然，在进行领域模型设计时，需要正确地甄别不同设计要素在设计模型中扮演的角色。正如《领域模型驱动设计》给出的角色构造型所示：



43487355.png

聚合内的实体与值对象负责处理与自身信息相关的领域行为，工厂和资源库负责管理聚合的生命周期，网关负责封装对外部资源的访问，而领域服务则封装了上述对象角色之间的协作，并被定义为类或接口，对外体现了一种领域行为。因此，领域服务应满足如下三个特征的任何一个：

* 领域行为与状态无关
* 领域行为需要多个聚合参与协作，目的是使用聚合内的实体和值对象编排业务逻辑
* 领域行为需要与访问包括数据库在内的外部资源协作

领域服务并非灵丹妙药，切忌将所有的领域逻辑都往领域服务塞，这也是为何要求领域服务的名称必须包含一个动词的原因。 **和谐的协作机制** 是好的面向对象设计，当领域服务对外承担了业务场景的领域行为时，要注意将不同的职责分配给不同的对象角色，尤其应遵循“信息专家模式”将数据与行为封装在一起，放到持有数据的聚合内对象中，再以行为的方式进行协作，保证职责分配的合理均衡。

下一章

互动评论



评论



王岩6 天前

有一个异议 ValidatingOrderService 这个验证是不是订单的领域能力我觉得需要更多的上下文 从文中来看，感觉对于客户是否存在的校验不属于订单的领域逻辑一部分；更多的是防御系统的消费者没有正确传递客户信息。按照这个思路和迪米特法则，我觉得这个验证是协调者的职责的一部分，就是应用层的职责，具体判定约束（给个id是否有对应的客户）应该放在客户上下文呢中，遵循询问而非告知定位原则 个人浅见，请张逸老师和各位同学批评

[鼓掌回复](javascript:void(0))

评论

查看更多

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持